In the Specification:

The title of the invention was objected because the trademark or trade name JAVA should be accompanied with the appropriate designation symbol. Please amend the title to read as follows:

Marshaling And Un-Marshaling Data Types In Xml And JAVA®

The specification was objected to because use of the trademark, JAVA, was not accompanied by a designation symbol. Applicant has corrected this by accompanying each letter of the word JAVA with a designation symbol ®.

Please replace paragraphs **[0009]; [0010]; [0026]-[0028]; [0030] – [0033]; [0036]; [0038]-[0039]; [0041]; [0043-0045]; [0050]-0056]; [0058]-[0059]; [0061]; [0064]; [0065] and [0069],** with the new paragraphs **[0009]; [0010]; [0026]- [0028]; [0030] – [0033]; [0036]; [0038]-[0039]; [0041]; [0043-0045]; [0050]-0056]; [0058]-[0059]; [0061]; [0064]; [0065] and [0069]** as below.

[0009] Certain drawbacks exist in the way data is currently transformed, or marshaled, between data types. In existing systems, a user starts with an existing Java type and asks the system to generate the XML schema that reflects the JAVA® type, and further to marshal the JAVA® data to the XML that was automatically generated. Most products that marshal XML run through a compiler, such as a JAVA® to WSDL compiler, in order to generate an XML schema. One drawback to such an approach is that only the scenario going from JAVA® to XML is addressed. Current tools are not particularly good at taking an existing XML schema and turning that entire schema into a convenient-to-use JAVA® type.

[0010] Another problem with current marshaling technologies appears when a user simply wishes to look at a small piece of XML data. That user may prefer to simply pass on the rest of

the XML data without processing that data. Current marshaling technologies are not effective at simply passing on the remainder of the data. Typically, going from marshaling to unmarshaling is complicated, as not all semantics in XML can be easily captured in JAVA®. If a user brings in a message, changes a small portion of the message, and tries to resend the message as XML, portions other than that changed by the user will be different, such that a lot of other information may be lost. If the XML contains wildcard attributes or elements, for example, those wildcards will not be retained. Information about element order may also be lost or scrambled, which is a problem if the schema is sensitive to element order.

[0026] Systems and methods in accordance with one embodiment of the present invention overcome many deficiencies in existing marshaling and unmarshaling systems by translating XML schemas, which define XML data in an XML document, into XML types in JAVA® when marshaling data between XML and JAVA®. XML types are actually JAVA® types which, in addition to regular JAVA® bean functions and access to database, can also access and update XML data within JAVA® in an efficient, type-safe, robust, and convenient way. An architecture can be used in at least one embodiment that provides the ability to handle almost 100% of the schema introduced by a user.

[0027] The use of XML types can allow the combination of XML- and JAVA® -type systems. This can be done in a way that allows developers to achieve loose coupling. XML schemas realized as XML types can remain fully faithful to the XML. It can be easy for a developer to take control of precise transformations between existing JAVA® types and existing XML types. XML types can address the JAVA®/XML boundary by bringing together several technologies, including for example schema-aware strongly typed access to XML data document from JAVA®, compact and indexed in-memory XML store, speedy and minimal (pull) parsing and (binary) serialization, lightweight document-cursor traversal, XPath and XQuery navigation and transformation, and seamless JAVA® IDE integration.

[0028] For an XML-oriented "XML to XML via JAVA® code" example, **Figure 1** shows a simple exemplary XML schema definition (XSD) file that can be used with embodiments of the present invention. This particular XML schema describes the type of a purchase order. The schema can be a pre-existing file that was generated by a schema tool or created by a user. In this example, it may be necessary to "clean up" or "fix" invalid XML purchase order information.

[0030] In order to write a program using XML types, an XML schema file can be added to a JAVA® project. An example of a system for XML marshaling and unmarshaling is shown in **Figure 2**. In one embodiment, the system can process the file with a compiler **100** that knows not only how to compile a JAVA® project **101**, but also is capable of compiling the XSD file **102.** When XSD file is compiled, a number of XML types **103** can be generated in addition to regular JAVA® types **104.** These XML types can then be added to the classpath. For example, an XML type called "purchase order" can be generated from the schema type called "purchase-order."

[0031] A JAVA® source code representation of the XML types compiled from the example schema in **Figure 1** is shown in **Figure 3**, where the source that generates these JAVA® types is the schema file itself. A type called LineItem corresponds to the line item-element nested inside the purchase-order element in the XSD file. In JAVA®, the XML line-item element can turn into something similar. For each element inside a type, a JAVA® field can be generated. For the "desc" element in the XSD, for instance, there are corresponding "getDesc" and "setDesc" methods in the generated type for each line item. For the purchase order as a whole, a user can obtain or send an individual line item. The names of the generated types can be automatically derived from the schema names. Each generated type can be annotated with the relevant schema name from which it comes. Each type can also extend an existing XML type.

[0032] In one embodiment, XML types can implement a common base XML type called "XMLObject". Such an XML type provides the ability to execute a number of XML-oriented data manipulations and will be referred to herein as an "XBean". An XBean is not a standard JAVA® bean, as an XBean inherits from an XMLObject. An XMLObject is unusual, in that an XMLObject as an XML type provides a way for each XBean to retrieve its original, or corresponding, XML. An XBean can be thought of as a design pattern for JAVA® types representing data, such as business data, that can be serialized and de-serialized to XML, as well as accessed in a type-safe way from JAVA®. XBeans can also be thought of as a small set of natural language idioms, either annotated JAVA® or non- JAVA®, for generating those types. Normally, there is a tradeoff when an application developer or component developer decides how to represent business data. If the data is represented as type-safe JAVA®, then serialization to XML or to databases can be awkward. If the data is represented as XML, then conversion to JAVA® types can also be somewhat awkward. The same holds true for conversion to either of the other types if data is in a result set from a database. It is therefore advantageous to provide a single category of JAVA® types that is convenient for passing, using, and manipulating as both XML and JAVA®. It is further advantageous that the same type is convenient for database access as well as form input and validation.

[0033] As Shown in **Figure 2**, XBeans in one embodiment can sit at the intersection of three types of business data: XML data **105**, JAVA® data **106**, and database **107.** An XBean can simply contain data, without any logic or means for communication or computation. An XBean can be defined using any of a number of idioms. Once an XBean is defined, the XBean can be used in various contexts. For example, an XBean can be defined using XML schema, then instantiated by attaching the XBean to an XML input stream. In this case, the JAVA® types that are generated can be passed around and used as a JAVA® bean, with friendly getters and setters.

[0036] Certain methods can determine what the XML looks like at any point in time. For example, XML types can be used to implement a Web service that executed the requested operation such as shown in **Figure 5**. The exemplary code in this Figure can be used to fix quantities in an entire purchase order. On lines 1-2 of the Figure, the method is declared as a Web service operation that takes a PurchaseOrder XML type. The Web services runtime can recognize XML types and pass incoming messages efficiently, without fully parsing the XML. The other XML type that was declared in the schema is LineItem, which can be seen on lines 4, 5, and 7 as an array called emptyItems. Since the compiled XML types are strongly typed, they allow validation of both schema, such as validating PurchaseOrder against a schema file, and JAVA® compiler checking, such as verifying that the type of the argument of emptyItems[i].setQty(1) is an integer.

[0038] Strongly typed JAVA® accessors may not be appropriate for all XML usage. In one embodiment, XML types can extend a base XML-oriented XMLObject type that provides, for example, XPath, XQuery, XMLCursor, and XMLReader. On line 5, the XMLObject getAllValues method executes an XPath on the input to locate all line item elements with qty=0. On lines 6 and 7, it can be seen that the LineItem types can be used to update the XML data document. Each instance of the type refers to a specific node in the document, and when methods such as setQty(1) are called, the data of the document are being manipulated in an easy, type-safe way. On line 8, the type is returned directly from the Web service to complete the function and send the response message.

[0039] Systems and methods in accordance with embodiments of the present invention can also deal with transformation among different XML types, where a user may need to process an XBean to retrieve data. For example, it may be necessary to clean up the line items by modifying the description and price to match the item ID. This can be done in one example by looking up

each catalog item in an existing application database. This work can be done using a JAVA® lookup method that can take an integer item id and return a CatalogItem type. For instance:

```
CatalogItem findCatalogItem(int catalogID);
class CatalogItem
{
  int getCatalogID();
  String getDescription();
  float getPrice();
}
```

This class appears to be similar to a LineItem XML type, but has some relatively minor differences. For instance, CatalogItem has no quantity and the item ID is called a "catalogID" rather than an "itemID." In fact, since CatalogItem is so similar to LineItem, it may be desirable in some situations to write code such as that shown in **Figure 6**. In the Figure the value of a complex XML type, LineItems, is being set to a complex ~~Java~~ JAVA type, CatalogItem. Each XML type has a set method that can take an arbitrary type, such that the above code can compile and run. At runtime, however, the user can get an XML conversion exception, complaining "No Transformation has been defined mapping from CatalogItem to LineItem". In such case, the user can select the appropriate CatalogItem type name and check the corresponding XML transformation. If a user has an existing bean that looks up the title of an item and returns a ~~Java~~ JAVA type called 'CatalogItem,' CatalogItem will have a CatalogID description that looks like the line items, but is slightly different than the prior example. For example, there is no quantity field, and what was previously called "ItemID" is now called CatalogID.

[0041] An example of a default type declaration for the CatalogItem class is shown in **Figure 7**. It can be visually written by XQuery into a file in a directory that contains all the transformations. An XQuery can be defined that tells the system how to transform a catalog item into a regular line item. An example of such a query is shown in **Figure 8**. The bolded text in **Figure 8** corresponds to an XQuery that maps catalogID into itemId and sets the extra field qty to 0. The remainder is an envelope that holds the XQuery in an XML file. This 0 value can be corrected, such as can be done in JAVA® as shown in **Figure 9**. On line 13 in **Figure 9**,

exemplary code can be seen where the catalog item is being sent into the items line. It can be desirable to take a catalog item that a user is getting from an existing API and get that item into an XBean that will represent a line item. Such code would not normally work, as the catalog item is different from the line item.

[0043] An alternative way to ensure that the quantity is correct is to define the CatalogItem through a line-item transformation to take two input arguments, such as a CatalogItem and an integer quantity. From this example, it can be seen that there is a global registry of transformations, indicating source types and target types. Sources and targets are allowed to be ~~Java~~ JAVA types. Whenever an automatic translation between two different types is required, the registry can be consulted. A registry can be used that allows a single JAVA® type to map to any number of different XML types, depending on the situation. A registry can also have the advantage that every mapping between any two given types need only be defined once, and then it is easily reusable.

[0044] In certain systems, difficulties may arise such that multiple versions of a schema may need to be dealt with at the same time in a single program. For this reason, there can be a provision for tagging each schema with a version identifier. The relevant JAVA® types and transformations can all be done separately, treating each version as its own type system.

[0045] In yet another example, a user may wish to write a Web service that takes a catalog item as input, or to expose an existing JAVA® function such as "findCatalogItem" as a Web service. For example, the following code could be written to expose findCatalogItem as a Web service in existing systems:

```
class MyWebService
{
  /** @jws:operation */
  CatalogItem findCatalogItem(int catalogID)
  {
```

```
        return MyDbUtilities.findCatalogItem(catalogID);
    }
}
```

[0050] Systems and methods in accordance with some embodiments can keep an XML schema and a corresponding JAVA® type in sync. A user with a strongly-typed JAVA® can begin to add new line items or to change quantities, for example. If that user then wants to run an XPath path on the JAVA® type, the Xpath may need to be run on the XML data document in the current form of the data. In this case, if a user makes a modification to a document, either on the XML side or on the strongly-typed JAVA® side, the appropriate portion of the other side can be invalidated. When a user subsequently looks at that other side, the previously-invalidated information can be faulted in.

[0051] In order to compile an XML schema, it can be necessary to parse the schema, or XSD file, which is referred to as "schema for schema", In other words, an XSD file that represents the appropriate schema for the XSD files themselves. If a system is supposed to be able to handle 100% of the schema passed to the system, and the system generates convenient JAVA® accessibility, it can be expected that the system uses its own generated types for understanding XSD files when the system reads schema. A user should be able to take the schema for schema and compile that into JAVA®, such that the system can simply use the JAVA®.

[0052] Systems and methods in accordance with some embodiments, as shown in **Figure 12**, there can be at least two parts to any piece of data, or XML data, including the legal type of the data **116** and the meaning of the raw data **119**. The legal type of the data defines what kind of data is regarded as valid for the current application. A schema can contain very specific details regarding the legal types of data, and can in some instances contain some detail regarding the meaning of the data. Once the legal type of the data is known, it is possible to generate an automatic type that provides access to that data in a strongly-typed way. This is possible in part because the schema can identify all the valid sub-fields of a given data field. It is then possible to

grant a user strongly-typed JAVA® access in the appropriate place(s). Even after a user has loaded the data and has the data in the appropriate type, the user may still not know what the data means. In such case, a schema compiler **117** can be used that understands the raw data. This is somewhat similar to what are known as compiler compilers, such as YACC ("Yet Another Compiler Compiler"), which are capable of taking an abstract grammar and compiling the abstract grammar into a syntax tree. Since XML is already a syntax tree, this is not a problem. XML is not, however, a constrained syntax tree. Any node can have a variety of elements beneath it in the tree. Once a user has a schema, which can be thought of as a grammar for XML, the user knows exactly what is supposed to be underneath any given node. Therefore, instead of using just a raw XML syntax tree, a user can take advantage of a schema-constrained syntax tree.

[0053] Systems and methods in accordance with one embodiment of the invention maintain each schema as a JAVA® type, including simple types. If a user has a schema that is a restriction of a simple type, it can be indicated in the schema. For instance, if a user-defined type to be an integer of a legal type, it has to be a five digit number between 10,000 - 99,000. It is not necessarily desirable to define this to be a simple integer type as in existing systems. Instead, the information can be generated into a JAVA® type. The name of the JAVA® type can then be generated from the schema, such as the name "restricted integer."

[0054] Another invariant that can be maintained by systems and methods in accordance with the present invention arises in cases where there are at least two types in a schema that are base types. If one of the types is a base type of the other, that relationship will connect the two types in JAVA®. A high-fidelity translation of typed systems can allow base types to be preserved.

[0055] A validation engine using complied XML type constraints **118** can also be used to allow a user to determine whether any relevant XML type **120** is valid according to the XML. For example, in XML a purchase order line item might have a description quantity, catalog number,

and a price. There may also be a restriction in the appropriate XML schema that indicates 'description' is optional, but 'catalogItemNumber' is not optional. In JAVA®, there is no way of indicating that a field is not optional, or cannot be null. As such, most people who do marshaling are not able to validate a bean. Validate methods in accordance with embodiments of the present invention can be used that allow a use to validate any bean against the XML type constraints, and to be informed of any validity problems.

[0056] In systems and methods in accordance with some embodiments, an XML type can be shared among multiple JAVA® components. An XBean can be automatically emitted, such as where an automatically generated XML type is defined for a user-defined component works. In such a case, XBeans representing parameters and return values can be auto-generated as inner classes to an XML control interface generated for the component. If the message types are actually shared across many components, it may not make sense to have private XBean types for each instance of the message. In such case, it should be possible to refer to an XBean type explicitly when defining a user-defined component, in order to explicitly control how the XML type of the component is shaped. For example:

```
package mypackage;

/**
 * @jws:xml-interface enable="true"
 */
class MyComponent
{
  /**
   * @jws:operation
   *   parameter-xml-type="MyData"
   *   return-xml-type="MyStringMessage"
   */
  String myOperation(String a, int i) { return a + i; }
}
```

By referencing the XBean type "MyData" in the component, such as for parameter-xml, it can be asserted that the bean has getters that correspond to the argument names. For example, getA

should return a String, and getI should return an int. If these types do not line up, it may result in a compile-time error. For return-xml, it can be asserted that the bean type is the return value, or that it has a single property whose type matches the return value. By referencing the XML type, the XML schema is being referenced that defines the type of input and output messages to this method. The schemas can be reusable since they have names such as "MyData". A map is also being referenced between the XML and the JAVA® types. The map can be attached to the MyData type as metadata and, since it is attached to a named type, the map can be reusable.

[0058] An XBean type can extend a base XML type, such that wherever XML can be passed, an XBean can be passed as well. In addition, any XBean can be attached to an XML data document, so wherever XML is available, an XBean can be created for convenient access to the data. In systems and methods in accordance with some embodiments, XBeans can be created easily and used in several different ways. For instance, an XBean can be created implicitly via the definition of a JWS (JAVA® Web services) method. An XBean can be created based on a parameter list of the function and the maps associated with the function. Also, an XBean can be created explicitly using a *.xbean file. A *.xbean file can have at least two different implementations, such as JAVA® Bean+Maps or XML+Query, each of which can freely use annotations. An example of implicitly creating a bean over a JWS operation might look like the following:

```
/**
 * @jws:operation
 * @jws:usebean beanName::creditCardInfo::
 **/
updateCreditCardInfo(String custId, String ccNumber, Date expDate, Address addr)
```

This would implicitly create an XBean using the default maps for the operation. Applying a specific map to the operation would create the XBean using those maps for input and output. If the beanName is already defined, the existing bean can be used. A separate syntax can be used when creating a bean, instead of using an existing syntax.

[0059] A simple JAVA® Bean+Maps *.xbean file might look like the following:

```
public interface xbean1

{
  /**
   * @xbean:property
   * */
  String name;
  /**
   * @xbean:property
   * */
  String address;
}
```

This would create a file with public get and set methods, as well as the standard XML that would be defined for this set of properties. A slightly more complex file would have a map attached, such as:

```
public interface xbean1 extends JAVA® XBean
{
  /**
   * @jws:inputxml xml-map::
   * <PERSON><NAME>{name}</NAME><ADDRESS>{address}</ADDRESS>::
   **/

  /**
   * @xbean:property
   * */
  String name;
  /**
   * @xbean:property
   * */
  String address;
}
```

The above examples use individual JAVA® members as native storage. Equally important can be the use of XML as a native storage. On the opposite side, a simple file could use XQuery to return values. This might look as follows:

```
public interface xbean1 extends XMLXBean
{
```

```
    private XML xml;

   void xbean1(XML xmlParam)
     {
     xml = xmlParam;
     }

   /**
    * @jws:xquery statement :: $xml/name/text() ::
    * */
   String getName();
}
```

[0061] Although a user can use and manipulate a strongly type such as PurchaseOrder as if it were an ordinary JAVA® type, behind the type can be an implementation that directly accesses and manipulates the underlying XML data. For example, immediately after a value is set in a strongly type, the same value can be available from any cursor that uses XPath to search the same set of data.

[0064] XML types can add the schema to the JAVA® runtime model. For example, every schema can compile into a JAVA® type at compile time. This can include both complex types and simple types. Precompiled types such as XmlString and XmlDate can be used for the fundamental and simple types built-in to XML Schema. XMLObject itself can correspond to the xsd:anyType. In addition, for each schema, a pointer resource can be generated into the target class hierarchy that provides a map from all schemas with a given name into corresponding ~~Java~~ JAVA® type names.

[0065] Multiple schemas can be allowed to have the same XML name, but different types with the same name may be tagged with different "XML world" names. Only one world may be allowed to be the default world. One way to control type generation is through an .xval file adjacent to the .xsd file at compile time. At runtime, indexed XML can be automatically schema-aware. The visibility of schemas can be tied to the current ClassLoader. A thread-local index of

visible schemas can be maintained. When a new schema is requested via fully-qualified XML name, a ClassLoader.getResourceAsStream call can be used to locate a pointer to the corresponding JAVA® type, such as in the default world. Lookups in a specific world can also be done. An implementation of XMLIndex can automatically resolve all XML to types using such a scheme. If no "xml world" is specified, a default world can be used. Other alternate views can also be specified that allow different versions of schemas to be used.

[0069] The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. Particularly, while the concept "type" is used for both XML and JAVA® in the embodiments of the systems and methods described above, it will be evident that such concept can be interchangeably used with equivalent concepts such as, interface, shape, class, object, bean, and other suitable concepts. Embodiments were chosen and described in order to best describe the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention, the various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.